Scalable Multiphysics Network Simulation Using PETSc DMNetwork

DANIEL A. MALDONADO, Argonne National Laboratory
SHRIRANG ABHYANKAR, Argonne National Laboratory
BARRY SMITH, Argonne National Laboratory
HONG ZHANG, Argonne National Laboratory

A scientific framework for simulations of large-scale networks, such as is required for the analysis of critical infrastructure interaction and interdependencies, is needed for applications on exascale computers. Such a framework must be able to manage heterogeneous physics and unstructured topology, and must be reusable. To this end we have developed DMNetwork, a class in PETSc that provides data and topology management and migration for network problems, along with multiphysics solvers to exploit the problem structure. It eases the application development cycle by providing the necessary infrastructure through simple abstractions to define and query the network. This paper presents the design of the DMNetwork, illustrates its user interface, and demonstrates its ability to solve large network problems through the numerical simulation of a water pipe network with more than 2 billion variables on extreme-scale computers using up to 30,000 processor cores.

Additional Key Words and Phrases: Networks, multiphysics, extreme-scale computing

1. INTRODUCTION

Modeling, simulation, and analysis of critical infrastructures—assets providing essential services that form the backbone of a nation's health, security, and economy including power distribution systems, water distribution, gas distribution, communications, and transportation—are of paramount importance from several strategically important perspectives, including maintaining sustainability, security, and resiliency and providing key insights for driving policy decisions. The Critical Infrastructures Act [Dominici 2001] defines critical infrastructures as "systems and assets, whether physical or virtual, so vital that the incapacity or destruction of such systems and assets would have a debilitating impact on security, national economic security, national public health or safety, or any combination of those matters." Such infrastructures comprise a large-scale network of assets spawned over wide geographical regions.

From the modeling and simulation standpoint, these critical infrastructures pose a multiphysics problem on an unstructured network with problem sizes from millions to billions of variables. Their analysis is computationally challenging because of the need for managing unstructured topology and heterogeneous node-edge characteristics. Several software packages assist scientists and engineers in modeling complex multiphysics networked problems (e.g., Simulink® [Mathworks 2017], Modelica [Association 2017], and LabView [Instruments 2017]), yet the problems they are able to solve are restricted by size. EPANET, a software for simulating water distribution piping systems [Rossman 2000] is a particular example of network simulation tool that can be implemented using our tools. For the scaling studies in this paper we incorporate a simple model of such a distribution system. Our work is not directly related to network analysis packages, —such as, SNAP [Leskovec and Sosič 2016], NetworkX [Hagberg et al. 2008], and NetworKit [Team 2017], —that do not provide simulation capability. Nor is it related to domain specific simulators such as the packet-level simulations modeling the transmission in a computer communication network in [Fujimoto et al. 2003] or the internet simulator NS-3 [Wehrle et al. 2010, p. 15–34].

Exascale computing provides opportunities and challenges for simulating such complex networked systems [Brase and Brown 2009]. In problems arising from fields as diverse as power distribution systems, water distribution, gas distribution, commu-

nications, and transportation, a common mathematical structure exists that can be exploited [Jansen and Tischendorf 2014] to develop scalable tools for multiphysics network simulation.

In this paper, we present a new package, DMNetwork, for modeling and simulation of multiphysics networks designed for exascale computers. DMNetwork provides the underlying infrastructure for managing the topology and the physics for large-scale networks. It is designed to scale for large networks while facilitating easy and rapid development of network applications. DMNetwork is seamlessly integrated in the *Portable Extensible Toolkit for Scientific Computing* (PETSc) [Balay et al. 2016], thus allowing the usage of the solvers available in PETSc. The name DMNetwork comes from the PETSc base class DM, that provides interfaces between the PETSc time-integrators and algebraic solvers and problem specific representations of mathematical models. For example, the PETSc DMDA class provides the connection between partial differential equation models on structured grids and the PETSc solvers.

2. NETWORKS, PHYSICS AND SOLVERS

Consider a series of physical elements (water pipes, gas pipes, power transmission lines) that are connected via junctions to form a network or a network of networks. Our paper presents abstractions that allow users to express their problem concisely, hide cumbersome data management operations, and use flexible and efficient solvers. The abstractions provide several convenient ways for networked system composition and decomposition, including the following.

(1) *Domain decomposition* [Smith and Tu 2013]: Decomposition of a computational domain into smaller subdomains. Each subnetwork is defined by a subregion of the entire region on which the original network is defined, as illustrated in Fig. 1.
(2) *Fieldsplit* [Brown et al. 2012; Smith et al. 2012]: Splitting of a multiphysics system into multiple single physics subsystems. Figure 2 shows a network composed of two subnetworks that represent distinct physics (e.g., electrical and water distribution).
(3) *Multilevel domain decomposition and fieldsplit*: Combination of domain decomposition and fieldsplit, as shown in Fig. 3.

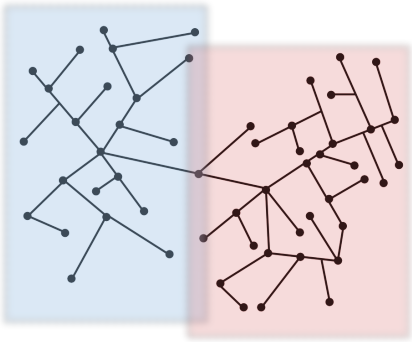

Fig. 1. A network is partitioned into two subnetworks, with regard to its topology.

.

In DMNetwork, to construct a network or a composite network, we add a series of physical elements, for example, water pipes, gas pipes, and power transmission lines as the edges of the graph, and pipe junctions, electrical generators and load systems

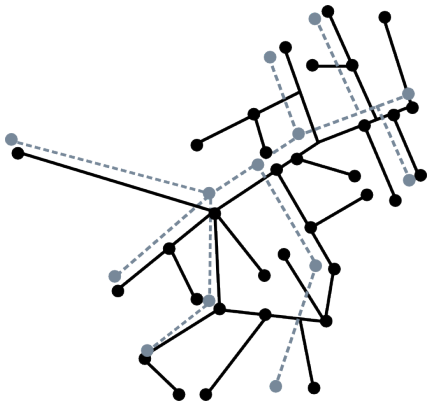

Fig. 2. A network, composed by two subnetworks of distinct physics (represented by solid and dashed lines) (e.g., electrical and water distribution). Customized solvers can be applied to the individual subnetwork.

.

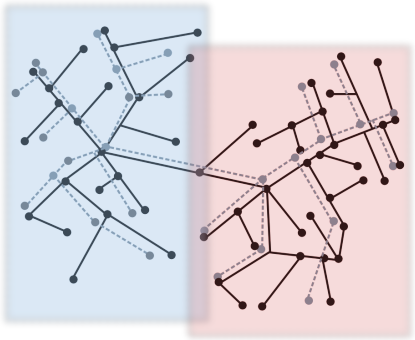

Fig. 3. A multilevel domain decomposition and fieldsplit solver can be used to solve this mixed case.

.

as the vertices of the graph. From this information DMNetwork builds a mathematical model for the entire physical system. The model is generally either a nonlinear algebraic equation or a differential algebraic equation (DAE). The Jacobian operator of such system often has the following structure:

$$N = \begin{bmatrix} P_1 & & & \dots & C_1 \\ & P_2 & & \dots & C_2 \\ & & P_3 & \dots & C_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ D_1 & D_2 & D_3 & \dots & J \end{bmatrix}, \tag{1}$$

where the submatrices P_i represent individual elements of the network: a water pipe or a power transmission line, a water reservoir or an electrical generator. These elements can have dissimilar structures and time scales. For instance, in power systems each vertex represents a mechanical generator that produces electrical energy, each of which will have different parameters and even different equations. In water networks, the edges represent water conduits modeled with partial differential equations, whereas the vertices represent either simple boundary conditions of pressure and flow

or complex mechanical regulators such as valves or variable height reservoirs. Furthermore, their physical meaning is independent from the network. The matrices J, C_i, and D_i contain information about boundary conditions of the individual systems system, such as continuity or energy conservation; that is, they provide constraints for systems P_i. They also provide information about the topological structure of the system (what is connected to what). The PETSc fieldsplit preconditioner can take advantage of this structure by employing customized preconditioners for each P_i.

Consider three coupled networks. We can write

$$N = \begin{bmatrix} N_1 & & & C_1 \\ & N_2 & & C_2 \\ & & N_3 & C_3 \\ D_1 & D_2 & D_3 & J \end{bmatrix}, \tag{2}$$

where J is the intersection network (e.g., overlapping elements in Fig. 1) and N_1, N_2, N_3 have the same structure as N in Equation (1). Domain decomposition preconditioners, such as block Jacobi and the overlapping additive Schwarz method [Balay et al. 2016], are amenable to parallel execution utilizing this structure. Depending on the network physics, it can be modeled by using a set of linear algebraic equations, a set of nonlinear equations, or a set of time-dependent equations. The PETSc library provides three layers of solvers: KSP, SNES, and TS, as illustrated in Table I, to solve these problems, respectively.

Table I. PETSc Solvers

Solver Name	Description	Mathematical Form
KSP and PC	Krylov subspace solvers and preconditioners	$Ax - b = 0$
SNES	Non-linear solvers	$F(x) = 0$
TS	Time stepping solvers	$F(t, x, \dot{x}) = 0$

DMNetwork is compatible with these solvers and provides a set of directives to help users set up their problem. As an example, consider a nonlinear system $F(x) = 0$. With Newton's method, we can find a sequence of approximations:

$$J(x_k)\Delta x_k = -F(x_k), \qquad x_{k+1} = x_k + \Delta x_k. \tag{3}$$

We obtain each iterate by evaluating $F(x_k)$ (residual function) and $J(x_k)$ (Jacobian) and solving Equation (3) iteratively to obtain the approximated solution x_k. For a large system comprising equations of different nature and different parameters, building the model requires the following tasks:

— Create problem specific data structures to distribute the parameters across the computer processors.
— Calculate the dimension of Jacobian matrix J, and preallocate J (determine the non-zero pattern of the sparse matrix).
— Evaluate the residual function and Jacobian for the given parameters.

With DMNetwork, the process is greatly simplified. The user needs to provide the number of degrees of freedom for each edge and vertex. Then DMNetwork takes care of the parallel distribution, preallocation, partitioning, and setting up of needed data structures to allow utilizing user-provided function evaluation routines. The user function merely needs to iterate by edges and vertices, retrieve the problem specific data and variables associated with them, and perform the local portion of the function evaluation based on their local mathematical models. PETSc also provides some tools to help approximate the Jacobian matrices efficiently via finite differences with coloring (discussed in the next section).

3. SOFTWARE STRUCTURE AND INTERFACE

In the preceding section we showed that the systems of equations arising from network problems present some mathematical structure amenable to parallelization. Furthermore, given that each subsystem of equations can present different physics or mathematical structure, supporting the use of different solvers for each subsystem is necessary. Since its inception, PETSc was meant to provide an environment for testing various solver options, particularly linear Krylov methods and the preconditioners they use, with minimal coding effort from the user. The user can often switch between solvers with runtime options.

DMPlex is a class in PETSc to handle general unstructured meshes, the mathematical models on the meshes and their connection to the PETSc solvers. [Lange et al. 2016]. DMNetwork is a recently developed subclass of DMPlex that provides specific abstractions for general networks. Both the vertices and edges can represent any number of physical models. New vertices and edges can be easily inserted through an API, and the existing ones can be removed or updated with minimum local changes. On multiple processors, DMNetwork can partition the network using the graph partitioning packages, such as ParMetis [Karypis et al. 2005] or Chaco [Hendrickson and Leland 1995], and distribute the problem specific data describing the physics to the appropriate processors.

3.1. DMNetwork Object and Its Interface

The DMNetwork object contains topological information about the problem as well as physical modeling data. In this section, we illustrate the DMNetwork interface. First we create a DMNetwork object. Users create their own problem specific data structures, based on C structs, that contain data associated to the physics occurring in the vertices and edges (for example, in the edge data structure, the user may include the electrical resistance of the circuit). These problem specific data structures are registered as components that can be accessed later via DMNetwork.

```
PetscInt VERTEX, EDGE; /* keys for the two models */
typedef struct _vertex Vertex;
typedef struct _edge Edge;
DMNetworkCreate(PETSC_COMM_WORLD,&dmnetwork);
DMNetworkRegisterComponent(dmnetwork,"vertex",sizeof(Vertex),&VERTEX);
DMNetworkRegisterComponent(dmnetwork,"edge",sizeof(Edge),&EDGE);
```

In this simple example, we have only two problem specific data structures, one for edges and one for vertices. In realistic simulations there are almost always more than two types.

Next, we provide the DMNetwork with information about the topology of the problem: the local number of vertices, the number of edges, and a list of the connectivity of the edges.

```
DMNetworkSetSizes(dmnetwork,nvertex,nedge,PETSC_DETERMINE,PETSC_DETERMINE);
DMNetworkSetEdgeList(dmnetwork,edgelist);
DMNetworkLayoutSetUp(dmnetwork);
```

In the final function call the DMNetwork creates a preliminary internal representation of the graph defined by the network. Edges and vertices each are assigned to a unique process. We can access the range of edges and vertices on the process, and add the problem-specific data parameters and the number of degrees of freedom for each entity.

```
DMNetworkGetEdgeRange(dmnetwork,&eStart,&eEnd);
for (i = eStart; i < eEnd; i++) {
  DMNetworkAddComponent(dmnetwork,i,EDGE,&edge[i-eStart])
  DMNetworkAddNumVariables(dmnetwork,i,nvar);
}
```

```
DMNetworkGetVertexRange(dmnetwork,&vStart,&vEnd);
for (i = vStart; i < vEnd; i++) {
  DMNetworkAddComponent(dmnetwork,i,VERTEX,&vertex[i-vStart]);
  DMNetworkAddNumVariables(dmnetwork,i,nvar)
}
```

The network then is partitioned and distributed to multiple processors to approximately equalize the number of unknowns per process.

```
DMSetUp(dmnetwork);
DMNetworkDistribute(&dmnetwork,0);
```

With these steps the user has created a distributed network object that contains the following:

— A (partitioned) graph representation of the problem
— A data structure containing needed ghost vertices and communication data structures needed to update the ghost values
— Memory space for the unknowns in each vertex and edge
— Physical data in each vertex and edge, related to the model for that entity
— Preallocation of the (Jacobian) matrices, that is, the determination of the nonzero structure of the matrix based on the graph.

Whether the user wants to solve the problem using a linear solver, a nonlinear solver, or a time-integrator, this DMNetwork object facilitates both the evaluation of the residual function (or right-hand side vector for linear problems), and computation of the (Jacobian) matrix.

3.2. A Example: Electric Circuit

We will solve a toy linear electric circuit problem from [Strang 2007]. The topology of the electrical circuit is shown in Fig. 4. The circuit must obey the Kirchhoff laws. Hence, in the vertices of the graph, the energy is not accumulated:

$$\sum_j i^{(j)} - i_{source(k)} = 0\,, \tag{4}$$

where $i^{(j)}$ is the current flowing though the branch (edge) j, incident to the node k. The $i_{source(k)}$ allows one to account for current sources at node k. The voltage drop across the edge k, from $v_{(i)}$ to $v_{(j)}$, is defined by Ohm's law plus any existing voltage source $v^{(k)}_{source}$:

$$\frac{i^{(k)}}{r^{(k)}} + v_{(j)} - v_{(i)} - v^{(k)}_{source} = 0\,. \tag{5}$$

We use the superscript and subscript to distinguish between edge and vertex quantities, respectively. In this case $i^{(*)}$ is an edge variable, and $v_{(*)}$ is a vertex variable.

These equations, as Strang[2007] shows, can be represented with the KKT matrix and the graph Laplacian. This structure is shared by many network flow problems, such as water networks:

$$\begin{bmatrix} R^{-1} & A \\ A^T & \end{bmatrix} \begin{bmatrix} i \\ v \end{bmatrix} = \begin{bmatrix} v_{source} \\ i_{source} \end{bmatrix}\,. \tag{6}$$

The practical implementation of this problem requires knowing the topology or connectivity of the network (a list of vertices and a list of edges defined by vertex pairs) and the physics (resistance, values of voltage source, and current source).

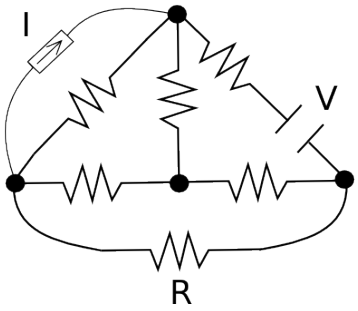

Fig. 4. Network diagram. Resistances are represented by zig-zag lines (R), voltage sources by parallel lines (V), and current sources by an arrow (I). The vertices of the graph are potentials, and the current flows across the edges.

.

3.3. Residual Function

The following is a code fragment from petsc/src/ksp/ksp/examples/tutorials/network/ex1.c [Balay et al. 2016]. It illustrates how to access DMNetwork information, iterate over the graph structure, and define the numerical problem that must be solved by computing the matrix and right-hand side vector for the linear system.

```
PetscErrorCode FormOperator(DM dmnetwork, Mat A, Vec b)
{
 DMNetworkGetComponentDataArray(dmnetwork,&arr);
 DMNetworkGetEdgeRange(dmnetwork,&eStart,&eEnd);
 DMNetworkGetVertexRange(dmnetwork,&vStart,&vEnd);
```

The array `arr` contains the physical data we provided in the `DMNetworkAddComponent()`. In order to achieve high performance, this problem specific data is stored and managed by DMNetwork in a simple one-dimensional array, as opposed to using pointers to a collection of C structs or C++ classes. Thus we access the correct data for a given entity through a variable offset that DMNetwork provides (shown below). We considered using a higher-level iterator construct to loop over the entities; but based on many years of experience with unstructured mesh codes, we have concluded that simple, basic data structures provide higher performance and a simplicity of debugging that is more important than the slightly simpler code syntax one can obtain with iterators. Of course developers are free to layer an iterator abstraction of their liking directly on top of the DMNetwork interface if they value the syntax it offers.

The construction of the matrix and right-hand side vector can be implemented with the following code:

```
for (e = eStart; e < eEnd; e++) {
  DMNetworkGetComponentTypeOffset(dmnetwork,e,0,NULL,&compoffset);
  DMNetworkGetVariableOffset(dmnetwork,e,&lofst);
  DMNetworkGetConnectedNodes(dmnetwork,e,&cone);
  DMNetworkGetVariableOffset(dmnetwork,cone[0],&lofst_fr);
  DMNetworkGetVariableOffset(dmnetwork,cone[1],&lofst_to);

  branch = (Branch*)(arr + compoffset);
  barr[lofst] = branch->bat; /* battery value */

  row[0] = lofst;
  col[0] = lofst;     val[0] =  1;
  col[1] = lofst_to;  val[1] =  1;
  col[2] = lofst_fr;  val[2] = -1;
```

```
15    MatSetValuesLocal(A,1,row,3,col,val,ADD_VALUES);
16
17    /* from node */
18    DMNetworkGetComponentTypeOffset(dmnetwork,cone[0],0,NULL,&compoffset);
19    node = (Node*)(arr + compoffset);
20
21    if (!node->gr) { /* not a boundary node */
22      row[0] = lofst_fr;
23      col[0] = lofst;   val[0] = -1;
24      MatSetValuesLocal(A,1,row,1,col,val,ADD_VALUES);
25    }
26
27     /* to node */
28     DMNetworkGetComponentTypeOffset(dmnetwork,cone[1],0,NULL,&compoffset);
29     node = (Node*)(arr + compoffset);
30
31     if (!node->gr) { /* not a boundary node */
32       row[0] = lofst_to;
33       col[0] = lofst;   val[0] = 1;
34       MatSetValuesLocal(A,1,row,1,col,val,ADD_VALUES);
35     }
36   }
37   for (v = vStart; v < vEnd; v++) {
38     DMNetworkIsGhostVertex(dmnetwork,v,&ghost);
39     if (!ghost) {
40       DMNetworkGetComponentTypeOffset(dmnetwork,v,0,NULL,&compoffset);
41       DMNetworkGetVariableOffset(dmnetwork,v,&lofst);
42       node = (Node*)(arr + compoffset);
43
44       if (node->gr) {
45         row[0] = lofst;
46         col[0] = lofst;   val[0] =  1;
47         MatSetValuesLocal(A,1,row,1,col,val,ADD_VALUES);
48       } else {
49         barr[lofst] += node->inj;
50       }
51     }
52   }
```

First we iterate over the edges (Line 1). For each edge we retrieve the component offset (that is, a pointer to the problem specific data for this edge), the edge variable offset, and the offsets for the variables in the boundary vertices (Lines 2–6). Next we write the Kirchhoff voltage law (Lines 11–15); and then, for each boundary vertex, we check whether the vertex is not a ghost value and write its contribution to the Kirchhoff current law (Lines 18–36). We then iterate over each vertex and add the contribution of each current injection to the corresponding equation.

3.4. Finite-Difference Jacobian Approximation with Coloring

In the engineering fields, from which many of the network problems arise, the models often have a complicated structure: they may include control logic and react in a discrete way to transients in the network. Writing an analytical Jacobian matrix evaluation subroutine is a daunting, time-consuming, and error-prone task. PETSc offers tools to calculate a finite-difference approximation of the Jacobian matrix suitable for some classes of problems. DMNetwork contains information about the connectivity of the vertices and edges, which enables building a sparse Jacobian matrix structure and using matrix coloring schemes [Coleman and More 1983] for efficient Jacobian evaluation. We have designed and implemented a customized, scalable matrix coloring routine for networks that minimizes the number of colors required and thus only incurs a small amount of interprocessor data communication. We also allow the input

of problem-specific information to reduce memory needs by using user-provided sparse matrix nonzero structures for individual edges and vertices:

```
1 DMNetworkEdgeSetMatrix(dmnetwork, e, Juser);
2 DMNetworkVertexSetMatrix(dmnetwork, v, Juser);
```

The finite-difference Jacobian approximation with coloring for networked applications then involves the following steps:

(1) User provides subroutine for local function evaluation.
(2) User provides the problem specific sparse matrix nonzero structures for the individual edges and vertices; if none are provided, dense matrix subblocks are used.
(3) DMNetwork builds the global sparse Jacobian matrix structure based on the global network topology and problem specific sparse matrix nonzero structures (when provided).
(4) Jacobian matrix is computed by finite-difference approximation using a matrix coloring scheme in an efficient and scalable fashion.

We present numerical experiments in Sec. 4.4 to demonstrate that this Jacobian approximation can be highly efficient and scalable on extreme-scale computers.

4. HYDRAULIC TRANSIENT SIMULATION ON EXTREME-SCALE COMPUTERS

Hydraulic transient simulations are performed for problems such as water distribution, oil distribution, and hydraulic generation. In this paper we focus on the simulation of water transients on closed conduits such as an urban distribution system. Simulation of hydraulic transients involves the calculation of pressure changes induced by a sudden change of velocity of the fluid [Chaudhry 1979; Wylie and Streeter 1978]. These velocity changes create a pressure wave that propagates proportionally to the speed of sound in the fluid media and the friction of the conduits. The modeling of this problem involves the solution of a set of PDEs. The disturbance of the system is introduced through a perturbation on the boundary conditions and results in a stiff differential equation, traditionally solved through the method of characteristics.

4.1. Description of the Problem

To facilitate discussion, we introduce the following notation:

nv, ne: number of junctions (vertices) and pipes (edges) of the network;
Q^k, H^k: water flow and pressure for pipe k;
Q_i, H_i: boundary values of water flow and pressure adjacent to junction i;
ne_i: number of connected pipes at junction i.

For pipe k, the water flow and pressure are described by the momentum and continuity equations:

$$\frac{\partial Q^k}{\partial t} + gA\frac{\partial H^k}{\partial x} + RQ^k\left|Q^k\right| = 0\,, \tag{7}$$

$$gA\frac{\partial H^k}{\partial t} + a^2\frac{\partial Q^k}{\partial x} = 0\,, \tag{8}$$

where g is the gravity constant, A is the area of the conduit, $R = \frac{f}{2DA}$ with f being the friction of the conduit and D its diameter, and a the velocity of the pressure wave in

the conduit. At an interior junction i, $ne_i > 1$, the boundary conditions satisfy

$$\sum_{j=1}^{ne_i} Q_i^{k_j} = 0\,, \tag{9}$$

$$H_i^{k_j} - H_i^{k_1} = 0, \quad j = 2 \ldots ne_i\,. \tag{10}$$

Equations (7)–(10) are built over a network or a network of subnetworks. Note that the physical meaning of these equations corresponds to conservation of energy and mass. Special boundaries such as the connection to a reservoir, valve, or pump provide application-specific boundary conditions.

4.2. Steady State

Systems engineers customarily divide the simulation of a dynamic system into two stages: steady state and transient state. Most systems are assumed to operate in steady-state conditions until some disturbance perturbs the system. In steady state the magnitudes do not vary with time. Thus Equations (7)–(8) are reduced to

$$gA\frac{\partial H^k}{\partial x} + RQ^k\left|Q^k\right| = 0\,, \tag{11}$$

$$\frac{\partial Q^k}{\partial x} = 0\,. \tag{12}$$

We can make two assertions for steady state: (1) along an individual pipe, the flow Q^k is constant; and (2) the drop of pressure H^k can be described with a linear function. Hence values (Q, H) inside a pipe can be uniquely determined by their boundary values, which satisfy the interior junction equations (9)–(10) and special boundary conditions. Equations (9) and (10) represent global connectivity of the network. Adding special boundary conditions, they form an algebraic differential subsystem that we call the *junction subsystem*. We name the rest of system the *pipe subsystem*. The junction subsystem has an ill-conditioned Jacobian matrix in general; its size is determined by the numbers of vertices, edges, and the network layout but is independent of the level of refinement used within the pipes for the differential equations (11)–(12). Thus it forms a small, but difficult to solve, subsystem requiring strong preconditioner for its solution.

The Newton-Krylov method [Kelley 2003] is used to solve the nonlinear steady state problem. For its linear iterations, we use the FieldSplit preconditioner to extract the junction equations from entire system, apply a direct linear solver (e.g., MUMPS parallel LU solver [Amestoy et al. 2001]), and use block Jacobi with ILU(0) in each subblock of the Jacobian for the rest of the system. The command-line options for this execution are as follows.

Preconditioner for entire system:

```
-initsol_pc_type fieldsplit
```

Preconditioner for junction subsystem:

```
-initsol_fieldsplit_junction_pc_type lu
-initsol_fieldsplit_junction_pc_factor_mat_solver_package mumps
```

Preconditioner for pipe subsystem:

```
-initsol_fieldsplit_pipe_pc_type bjacobi
```

```
-initsol_fieldsplit_pipe_sub_pc_type ilu
```

The prefix *initsol* is used to distinguish the steady-state system from the transient system, which is solved next. In Table II we compare the performance and strong scalability of the LU and FieldSplit preconditioners for a single network with 3,949,792 variables on Edison, a Cray XC30 system (see Sec. 4.4). Most of the time is spent in the LU factorization. Hence when we use FieldSplit employing block Jacobi on the larger subdomain, computation time is greatly reduced.

Table II. Comparison of LU and FieldSplit Preconditioners

No. of Cores	SNES Solution Time (sec)		KSP Solution Time (sec)	
	LU	FieldSplit	LU	FieldSplit
24	52.0	3.91	49.1	1.10
48	45.9	3.84	43.0	0.93
72	42.3	2.92	41.0	0.67

The steady state solution was compared with the EPANET software using a benchmark case from [de Corte and Sörensen 2014] consisting on 74 junctions, 102 pipes and one reservoir. The solution of the problem was consistent with EPANET's solution, minding the differences in the approach.

4.3. Transient State

In the transient state we calculate the pressure wave that arises after a perturbation is applied to the steady state solution. We solve Equations (7)-(10), which are a set of hyperbolic partial differential and algebraic equations. We have simulated a case appearing in [Wylie and Streeter 1978, p. 38] to benchmark the accuracy of our solution. The case consists on a single pipe connected to a reservoir and a valve. At time $t = 0^+$ the valve is closed instantaneously, creating a pressure wave that propagates through the pipe back and forward. In Fig. 5, we plot the pressure profiles (hydraulic or piezometric head, in water column meters) for a set of equally spaced points along the water pipe. In particular, the dark blue curve in the figure represents the pressure at the water reservoir. The water reservoir is treated as a constant pressure source, and hence the pressure is constant through time. On the other extreme, in light blue, we can see the pressure wave next to the valve, which sharply increases after its closure. The point right next to the valve (in yellow) will not be perturbed until the pressure wave has reached it, at a time that is proportional to the speed of the wave. This example gives us a physical intuition of the importance of the Courant number in computing hyperbolic PDEs. Several methods to solve such systems are employed in the literature. In this section we describe two of the most common ones.

4.3.1. Method of characteristics. The method of characteristics [Chaudhry 2014] involves applying a change of variables to Equations (7)–(8). Taking (8), scaling it by a term λ, and adding it to (7), we obtain

$$(\frac{\partial Q^k}{\partial t} + \lambda a^2 \frac{\partial Q^k}{\partial x}) + \lambda g A (\frac{\partial H^k}{\partial t} + \frac{1}{\lambda}\frac{\partial H^k}{\partial x}) + R Q^k \left|Q^k\right| = 0\,. \tag{13}$$

Using the chain rule, we obtain

$$\frac{dQ}{dt} = \frac{\partial Q}{\partial t} + \frac{\partial Q}{\partial x}\frac{dx}{dt}\,, \tag{14}$$

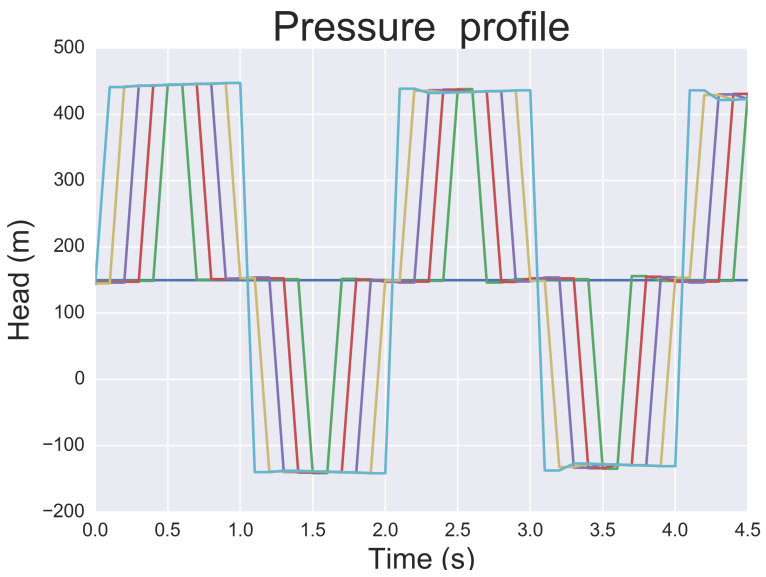

Fig. 5. Pressure wave on a conduit. This pressure wave is created in a single pipe with a conduit, when the end of the conduit is instantaneously closed. The vertical axis measures the piezometric head in water column meters, and each different color shows the pressure of a point in the pipe.

.

and

$$\frac{dH}{dt} = \frac{\partial H}{\partial t} + \frac{\partial H}{\partial x}\frac{dx}{dt}\,. \tag{15}$$

By defining

$$\frac{1}{\lambda} = \frac{dx}{dt} = \lambda a^2\,, \tag{16}$$

that is $\lambda = \pm\frac{1}{a}$, we obtain the ODEs in which the independent variable x has been eliminated:

$$\frac{\partial Q}{\partial t} \pm \frac{gA}{a}\frac{\partial H}{\partial t} + RQ\,|Q| = 0\,. \tag{17}$$

4.3.2. Lax scheme. An alternative to the method of characteristics is the Lax scheme [Chaudhry 2014], an explicit first-order scheme. We approximate the partial derivatives as

$$\frac{\partial H}{\partial t} = \frac{H_i^{j+1} - \bar{H}_i}{\Delta t}, \qquad \frac{\partial Q}{\partial t} = \frac{Q_i^{j+1} - \bar{Q}_i}{\Delta t}\,, \tag{18}$$

$$\frac{\partial H}{\partial x} = \frac{H_{i+1}^{j} - H_{i-1}^{j}}{2\Delta x}, \qquad \frac{\partial Q}{\partial x} = \frac{Q_{i+1}^{j} - Q_{i-1}^{j}}{2\Delta x}\,, \tag{19}$$

$$\bar{H}_i = 0.5(H_{i-1}^{j} + H_{i+1}^{j}), \qquad \bar{Q}_i = 0.5(Q_{i-1}^{j} + Q_{i+1}^{j})\,. \tag{20}$$

These equations are applicable only to interior points; hence we use the characteristic equations at the boundaries.

4.4. Experimental Results

We conducted experiments on two computer systems: Cetus, an IBM Blue Gene/Q supercomputer in the Argonne Leadership Computing Facility [ALCF 2017], and Edison,

a Cray XC30 system in the National Energy Research Scientific Computing Center [NERSC 2017]. Cetus has 4,096 nodes, each with 16 1600 MHz PowerPC A2 cores with 16 GB RAM per node, resulting in a total of 65,536 cores. Edison has 5,576 compute nodes. A node has 64 GB memory and two sockets, each with a 12-core Intel processor at 2.4 GHz, giving a total of 133,824 cores.

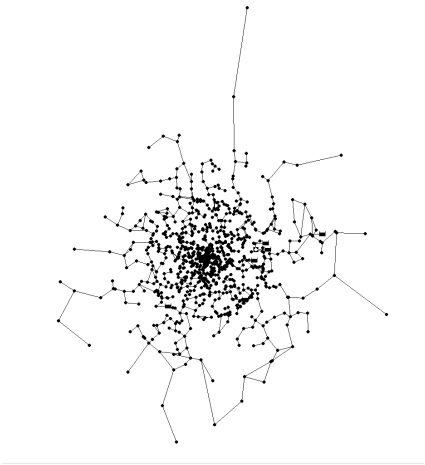

Fig. 6. Water network provided by [de Corte and Sörensen 2014].

.

Our test network was obtained from [de Corte and Sörensen 2014] in EPANET format [Rossman 2000]. It consists of 926 vertices and 1,109 edges of various lengths (see Fig. 6). Equations (7)–(10) were discretized via the finite volume method into a system over this single network with a total of 3,949,792 variables. We call the resulting DMNetwork *sub-dmnetwork*.

We then built a large network by duplicating *sub-dmnetwork* and composing them into a composite dmnetwork. Artificial edges were added between the sub-dmnetworks without introducing new variables over these edges. To perform weak-scaling studies of the simulation when the number of sub-dmneworks was doubled, we increase the number of processor cores proportionally.

The focus of our simulation is the time integration of the transient-state system, with its initial solution computed from the steady-state system (11)–(12). All the experiments were done by using the PETSc DAE solver, that is, the Lax scheme for each pipe and backward Euler time integration for the entire composite network system (7)–(10). The Jacobian matrix was approximated by finite differencing with coloring, as discussed in Sec. 3.4, and the Krylov GMRES iterations with selected preconditioner were implemented for the linear solves. Since the Jacobian matrix structure does not change with the time, the computing time spent at each time step remains approximately constant. We run only 10 time steps for each test case.

Tables III and IV show the scalability of the simulation on Cetus and Edison. The linear solvers dominate the computation with efficiency determined by the selected preconditioners. Among all the preconditioners provided by PETSc, we found the block Jacobi and the additive Schwarz method (ASM) with subdomain overlapping 1 (ov.1) and 2 (ov.2) to be the most efficient for our application. In the tables, we compare these three preconditioners using the total simulation time and cumulative number of linear iterations (given in parentheses).

Table III. Execution Time of Transient State on Cetus

No. of Cores	Variables (in millions)	Linear Preconditioner		
		Block Jacobi	ASM ov. 1	ASM ov. 2
256	16	60.0 (43)	50.5 (24)	45.3 (20)
1,024	63	63.4 (49)	50.6 (24)	45.4 (20)
4,096	253	86.1 (54)	72.8 (34)	58.7 (20)
16,384*	1,012	94.1 (54)	81.2 (34)	65.3 (20)

* We set the number of cores per node to 8 (instead of 16) to double the memory available per core.

Table IV. Execution Time of Transient State on Edison

No. of Cores	Variables (in millions)	Maximum Variables per Core (in thousands)	Linear Preconditioner		
			Block Jacobi	ASM ov. 1	ASM ov. 2
240	16	106	9.9 (48)	7.3 (25)	6.4 (20)
960	63	106	10.6 (55)	7.0 (24)	6.2 (20)
3,840	253	106	10.4 (53)	7.3 (24)	6.7 (20)
15,360	1,012	104	11.9 (53)	11.4 (26)	9.9 (20)
30,720	2,023	117	20.0 (53)	17.6 (26)	17.2 (20)

The water pipe network has pipes of various lengths which gives rise to pipes with varying numbers of degrees of freedom. Since each pipe is restricted to a single process, some job imbalance results. As the number of cores increases, the job imbalance worsens, as indicated by the maximum number of variables per core in column 3 of Table IV. This imbalance affects the scaling with increased number of cores.

To investigate this, we then doubled the number of sub-dmnetworks for each test and listed the results in Table V. As the work pool increased for each core, the job balance as well as the scalability improved.

Table V. Execution Time of Transient State on Edison on Doubled Problem Sizes

No. of Cores	Variables (in millions)	Maximum Variables per Core (in thousands)	Linear Preconditioner		
			Block Jacobi	ASM ov. 1	ASM ov. 2
240	32	151	14.1 (40)	11.8 (24)	10.4 (20)
960	126	152	14.5 (47)	11.1 (24)	10.1 (20)
3,840	506	157	16.2 (50)	12.1 (24)	11.2 (20)
15,360	2,023	162	18.7 (50)	15.7 (24)	16.9 (20)

Table VI shows the weak scalability of the residual function evaluation and Jacobian evaluation using the matrix coloring scheme developed for the DMNetwork as described in Sec. 3.4. The data are taken from the cases using the block Jacobi preconditioner. Other cases give similar scalabilities. Along with the total execution time spent on these evaluations, we list their percentage of total time in the transient-state simulation.

In PETSc, the initial DMNetwork data structure representing the network is currently built by a single process sequentially; then it is distributed to the multiple processes for parallel construction of the physics and computation (see Sec. 3.1). Thus the size of application we can experiment with is limited by the local memory of computer systems. Cetus has 1 GB or 2 GB of memory per core when 16 cores or 8 cores are used per node, respectively; and Edison has 2.67 GB of memory per core. The largest

Table VI. Execution Time of Residual and Jacobian Evaluation on Edison

No. of Cores	Variables (in millions)	Residual Function	Jacobian Matrix
240	16	0.9 (7 %)	0.9 (9%)
960	63	0.9 (6 %)	1.0 (9 %)
3,840	253	0.9 (7 %)	1.0 (9 %)
15,360	1,012	1.4 (6 %)	1.5 (12 %)
30,720	2,023	2.8 (5 %)	3.1 (14 %)

problems we are able to run on these machines are approximately 1 billion variables using 16,384 cores on Cetus and 2 billion variables using 30,720 cores on Edison, respectively. Enhancements to DMPLex to support parallel construction of the graph will immediately provide this capability to DMNetwork.

The results demonstrate that, using DMNetwork, we can achieve satisfying weak scalability for the simulation with 2 billion variables using up to 30,000 cores. Overall, for most test cases, ASM, with an overlap of 2, is the fastest with the fewest number of linear iterations.

5. CONCLUSIONS

This paper introduces DMNetwork, a new class in PETSc for the simulation of large network problems. DMNetwork interfaces with all the solvers available in PETSc, providing the user with the ability to switch between different solvers with minimal effort and offering an effective test base for network structured problems. DMNetwork greatly simplifies programming parallel code to solve potentially complicated network problems. Large-scale experiments with a water network show the robustness of the data structures and the scalability of the data structures and solver.

ACKNOWLEDGMENTS

We thank Mathew Knepley for his help in designing and implementing DMNetwork. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

ALCF. 2017. Cetus supercomputer. http://www.alcf.anl.gov/cetus-and-vesta. (2017).

Patrick R. Amestoy, Ian S. Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.

Modelica Association. 2017. Modelica web page. (2017). https://www.modelica.org/

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2016. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.7. Argonne National Laboratory. http://www.mcs.anl.gov/petsc

James M. Brase and David L. Brown. 2009. Modeling, simulation and analysis of complex networked systems: A program plan. *Lawrence Livermore National Laboratory* May (2009).

Jed Brown, Matthew G. Knepley, David A. May, Lois C. McInnes, and Barry F. Smith. 2012. Composable linear solvers for multiphysics. In *Proceeedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*. IEEE Computer Society, 55–62. http://doi.ieeecomputersociety.org/10.1109/ISPDC.2012.16

M. Hanif Chaudhry. 1979. *Applied Hydraulic Transients*. Van Nostrand Reinhold Company.

M. Hanif Chaudhry. 2014. *Applied Hydraulic Transients*. Springer.

Thomas F. Coleman and Jorge J. More. 1983. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.* 20, 1 (Feb. 1983), 187–209.

Annelies de Corte and Kenneth Sörensen. 2014. HydroGen: an artificial water distribution network generator. *Water Resources Management* 28, 2 (2014), 333–350.

Pete V. Dominici. 2001. Critical Infrastructures Protection Act of 2001. (2001). https://www.gpo.gov/fdsys/pkg/BILLS-107s1407is/pdf/BILLS-107s1407is.pdf

Richard M. Fujimoto, Kalyan S. Perumalla, Andy Park, H. Wu, Mostafa H. Ammar, and George Riley. 2003. Large-scale network simulation: how big? how fast?. In *11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS 2003)*. IEEE, Orlando, FL, USA.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008), Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA)*. 11–15.

Bruce Hendrickson and Robert Leland. 1995. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*. ACM Press, New York, 28. DOI:http://dx.doi.org/10.1145/224170.224228

National Instruments. 2017. LabVIEW web page. (2017). http://www.ni.com/labview/

Lennart Jansen and Caren Tischendorf. 2014. A Unified (P)DAE Modeling Approach for Flow Networks. In *Progress in Differential-Algebraic Equations*, S. Schöps, A. Bartel, M. Günther, E. Maten, and P. Müller (Eds.). Springer, Berlin, Heidelberg, 127–152.

George Karypis et al. 2005. ParMETIS Web page. (2005). http://www.cs.umn.edu/~karypis/metis/parmetis.

C. T. Kelley. 2003. *Solving nonlinear equations with Newton's method*. SIAM.

Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. 2016. Efficient mesh management in Firedrake using PETSc-DMPlex. *SIAM Journal on Scientific Computing* 38, 5 (2016), S143–S155. http://epubs.siam.org/doi/abs/10.1137/15M1026092

Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

Mathworks. 2017. SIMULINK web page. (2017). https://www.mathworks.com/products/simulink.html

NERSC. 2017. Edison Supercomputer. https://www.nersc.gov/users/computational-systems/edison/. (2017).

Lewis A. Rossman. 2000. *EPANET 2 Users Manual*. Technical Report. Water Supply and Water Resources Division, National Risk Management Research Laboratory, Cincinnati, OH 45268.

Barry Smith, Lois Curfman McInnes, Emil Constantinescu, Mark Adams, Satish Balay, Jed Brown, Matthew Knepley, and Hong Zhang. 2012. *PETSc's software strategy for the design space of composable extreme-scale solvers*. Preprint ANL/MCS-P2059-0312. Argonne National Laboratory. DOE Exascale Research Conference, April 16-18, 2012, Portland, OR.

B. F. Smith and X. Tu. 2013. *Encyclopedia of Applied and Computational Mathematics*. Springer, Chapter Domain Decomposition.

Gilbert Strang. 2007. *Computational Science and Engineering*. Wellesley-Cambridge Press.

NetworKit Development Team. 2017. NetworKit web page. (2017). http://network-analysis.info

Klaus Wehrle, Mesut Güneş, and James Gross. 2010. *Modeling and Tools for Network Simulation*. Springer Berlin Heidelberg.

E. Benjamin Wylie and Victor L. Streeter. 1978. *Fluid Transients*. McGraw-Hill Book Company.